MIIC OPeNDAP Plugin

Version 1.6

The MIIC OPeNDAP plugin was designed as a set of general-purpose functions to extract averaged, filtered, and/or arbitrarily mutated data in useful formats. It works on any HDF, netCDF or other file that is supported by OPeNDAP. Currently the OPeNDAP Hyrax server version 1.13.2 is required.

The primary MIIC functions are tuple, compact, histo, and profile, which return averaged or filtered data. A suite of additional functions help you specify exactly the data variables, filters, algorithms, or other plugin behavior you need.

What's new?

NEWS

```
Version 1.6.1
* Fixes output of "compact" function -- return var dimensions are unchanged
(previous version would return "simplified"
  dimensions when dim_multiple was used)
* Rebuilt with Hyrax 1.13.2

Version 1.6
* Dynamic expressions stored locally in YAML file for security

Version 1.5.3 (15 Mar 2016)
* Fixes bug: functions customizing dimensions are applied in order so that
```

```
multiple changes can be made.
  For example, if 3072 => 1536x2 and 1536 => 192x8, then 3072 => 192x8x2
* Fixes tuple crash when returning a data variable that was never filtered
* Renames "histo" as "profile" and makes a real histogram function and data
type.
* Splits tuple with DIM_FORMAT into separate SSF and data type (filter).

Version 1.5.2 (13 Jan 2016)
* Fixes a memory leak
* Adds data value min/max properties to tuple
* Groups all tuples together by default, unles NO_TUPLE_MERGE keyword

Version 1.5.1 (5 Aug 2015)
* Fixes bug: histograms were not using dim_multiple when applying filters
to data vars

Version 1.5 (13 Jul 2015)
* Refactored histogram_2d_avg function into general "histo" function that
generates 1,2 or 3D histograms

Version 1.4 (1 Jul 2015)
* Added filter parameter REQUIES_DIMS to limit filters so that they only
apply to data variables with at least as many dimensions as the filter.
* Added function dim_multiple() to register a list of dimensions as being a
multiple of another list of dimensions.

Version 1.3.1 (29 Apr 2015)
* Adds support for dimension name aliasing.

Version 1.3.0 (15 Apr 2015)
* First stable release candidate for the MIIC OPeNDAP plugin.
* Includes refactored global filtering that doesn't require array copies.
* Also supports using OPeNDAP projections on arrays passed to SSFs.

Version 1.0.0 (19 Mar 2015)
* Provides generic 2D histograms and tuples
* Interface changes to generalize filters and data variables
* Performance improvements from previous version
```

```
Version 0.0.1 (01 Jan 2015)
* Pre-release versions supports histos on limited datasets and
spatial/spectral convolution using MODIS/SCIAMACHY only
```

Known Issues

Hyrax Data Server

Please be aware of these Hyrax issues which may cause some of your SSFs to fail.

Key	Summary	Type	Created	Updated	Due	Assignee	Reporter	Priority	Status	Resolution

⚠ JIRA project doesn't exist or you don't have permission to view it.
View these issues in JIRA

MIIC Plugin

Bugs in the MIIC Plugin itself.

Key	Summary	Type	Created	Updated	Due	Assignee	Reporter	Priority	Status	Resolution

⚠ JIRA project doesn't exist or you don't have permission to view it.
View these issues in JIRA

Calling Server-side Functions

The general format of an OPeNDAP server-side function (SSF) call is:

```
http://www.opendapserver.org:8080/path/to/datafile.nc?ssf(nestedssf(...),...)
```

- HTTP address of the OPeNDAP server and data file
- OPeNDAP response type: .nc (netcdf3), .ascii (ascii text), .dods (OPeNDAP binary format)
- Query part containing SSF calls, nested to any depth. No spaces.

Requests may be sent to the OPeNDAP server using HTTP GET or POST. OPeNDAP servers should be configured to accept POST, since SSFs may become too large for GET requests. If using POST, the body is the query part.

Since HTTP requests are client-initiated and synchronous, care must be taken by clients not to overload the OPeNDAP server. The OPeNDAP front-end (OLFS) configuration specifies how many simultaneous HTTP requests can be handled, by the number of back-end servers (BES) and client pool size(s). Don't exceed that number, or else your HTTP requests may start timing out.

Versioning

The built-in SSF *version()* will tell you what SSFs are available at a server and what version they are. Note that the file used to call *version* is irrelevant... you just need any old data file in order to invoke the SSF:

```
http://miic1.larc.nasa.gov:8080/opendap/CERES/SSF/Aqua-FM3-MODIS_Beta2-Ed3
/2004/01/CER_SSF_Aqua-FM3-MODIS_Beta2-Ed3_036042.2004010100.ascii?version()
```

The response is encoded XML. Clients can parse the response to see the deployed version of every registered SSF. For the MIIC Plugin functions, the version number is the plugin version when that function was last modified. For example, the function x_axis was last modified in plugin version 1.3, so its version is also 1.3. The function z_axis wasn't added until plugin version 1.5 so its version is also 1.5.

The overall MIIC plugin version is therefore the maximum of all individual function versions. For simplicity, this can also be obtained by calling the function *miic_version,* which was introduced in version 1.5:

```
<ds:function  name="miic_version" version="1.5" type="basetype"
role="http://something.about.roles.org/something.html" >
  <ds:Description
href="https://wiki.earthdata.nasa.gov/display/MIIC/MIIC+Plugin">Version of
MIIC plugin</ds:Description>
</ds:function>
```

```
http://miic1.larc.nasa.gov:8080/opendap/CERES/SSF/Aqua-FM3-MODIS_Beta2-Ed3
/2004/01/CER_SSF_Aqua-FM3-MODIS_Beta2-Ed3_036042.2004010100.ascii?miic_ver
sion()Return value: 1.5
```

Filtering

All functions returning data supports filters. Filters are always global in that they will remove observations from *all* data variables with compatible dimensions. For example, a range filter on latitude from 0 to 30 will discard the pink values from longitude below:
latitude[line][scan] – keep values from 0 to 30

23.1	23.2	23.1	27.1	27.4
27.9	31.1	34.2	34.2	34.1
34.8	34.8	34.9	34.8	34.9

longitude[line][scan] – as filtered by latitude

100.1	100.9	101.5	102.2	102.8
100.2	100.9	101.5	102.2	102.8
100.1	100.8	101.4	102.1	102.7

Multiple filters are combined such that an observation must *pass all filters* to appear in the output. This is a logical AND operation, as shown below. If you need to remove observations based on more complex criteria, see the function define_filter_mask.

pass filter1?	pass filter2?	in output?
Yes	Yes	Yes
No	Yes	No
Yes	No	No
No	No	No

Data Variable Dimensions

The shapes of data variables returned or used in filters ***do not need to match***, and there is no restriction on the number of filters or their dimensions. The plugin assumes that dimensions from all filters and data variables can be logically combined together. Two dimensions with the same name and size are assumed to be the same logical dimension. (The function dim_alias should be used in cases where the data product uses inconsistent dimension names).

When the filter has fewer dimensions than the data variable, we simply discard all of the extra-dimensional observations at one filtered index. For example, using a range filter on longitude[FOV] from 0 to 101:
longitude[FOV] – keep values from 0 to 101

100.1	100.9	101.5	102.2	102.8

temperature[FOV][band] – as filtered by longitude

8	7	23	4	5
7	31	34	34	3
5	34	34	34	1

When the filter variable has **more** dimensions than the data variable, the default behavior is to remove observations where **all values** would be filtered. For example, using a range filter on temperature[FOV][band] from 0 to 10, the pink cells will be removed:
temperature[FOV][band] – keep values from 0 to 10

8	7	23	4	5
7	31	34	34	3
5	34	34	34	1

longitude[FOV] – as filtered by temperature

100.1	100.9	101.5	102.2	102.8

In some cases however it may not make sense to apply a filter to a data variable even though they share a dimension in common. For example, consider a data product where two sets of data variables share a common dimension but are probably intended to be logically distinct.

- *latitude_type1[lines=1200][type1_scans=2000]*
- *longitude_type1[lines=1200][type1_scans=2000]*
- *temperature[lines=1200][type1_scans=2000]*

- *latitude_type2[lines=1200][type2_scans=3200]*
- *latitude_type2[lines=1200][type2_scans=3200]*
- *flux_capacity[lines=1200][type2_scans=3200]*

In this case only the "type1" lat/lons should be used to filter on *temperature,* and only the "type2" lat/lons should be used to filter on *flux_capacity.* One way to ensure this is to issue two separate OPeNDAP requests: one for temperature filtered by lat/lon "type1", another for flux_capacity filtered by lat/lon "type2".
This can also be achieved by using the keyword DIMS_REQUIRED when defining the filters on data variables *latitude_type1,* etc. That keyword indicates that the filter only applies to data variables with *at least* the same dimensions as the filter variable. (i.e. a filter on *latitude_type1* will apply to *temperature* but not *flux_capacity.)*

Overzealous Filtering
Because the default filtering behavior is to assume that all filter dimensions apply universally, this can result in requests that take a long time to finish.
In the above example, if the user creates filters for both sets of lat/lon variables and does **NOT** use the DIMS_REQUIRED keyword, the plugin must logically filter using an extremely large array:

- GLOBAL_FILTER[lines=1200][type1_scans=2000][type2_scans=3200]

Dimension Multiplicity

In some cases (e.g. VIIRS), data products include parameters that represent the same quantity at different resolutions. For example, consider a data product that represents temperature at twice the resolution of latitude & longitude:

- *latitide[scan=1200][pixel=1600]*
- *longitude[scan=1200][pixel=1600]*

- *temperature[scan_hi=2400][pixel_hi=3200]*

In this dataset, each latitude/longitude value corresponds to 4 temperature values. The function dim_multiple can be used to instruct the plugin that the dimensions [scan_hi][pixel_hi] are an even multiple of the dimensions [scan][pixel].

Filtering works much the same way as when combining dimensions together, where additional pseudo-dimensions are added to represent the multiple. For example, the parameter *temperature[scan_hi][pixel_hi]* will be treated logically by the plugin as *temperature[scan][2][pixel][2].*

When a data variable is an even multiple of the filter variable, the plugin discards all extra data observations at one filtered index. For example, using a range filter on longitude[scan][pixel] from 0 to 101, the pink cells are discarded:
longitude[scan][pixel] – keep values from 0 to 101

100.3	105.6	106.4	107.8	108.6
100.4	100.7	106.4	105.8	108.6
100.4	105.6	100.5	105.8	108.7
100.6	100.9	100.3	100.6	106.6

temperature[scan_hi][pixel_hi] – as filtered by longitude[scan][pixel] w/ dim_multple

8	5	6	12	62	6	7	23	4	5
7	7	8	5	33	22	31	34	34	3
5	1	12	1	32	1	34	34	34	1
3	4	7	4	5	12	2	8	7	12
11	5	7	21	6	11	7	1	3	12
41	12	4	6	1	9	27	32	7	5
20	11	3	8	1	51	7	7	27	7
10	3	5	3	6	6	7	9	22	6

As before, when the filter variable is an even multiple of the data variable, we only remove observations where all values would be filtered. For example, using a range filter on temperature[scan_hi][pixel_hi] from 0 to 10, the pink cells will be removed:
temperature[scan_hi][pixel_hi] – keep values from 0 to 10

8	5	6	12	62	6	7	23	4	5
7	7	8	5	33	22	31	34	34	3
5	1	12	1	32	1	***34***	***34***	34	1
3	4	7	4	5	12	***12***	***18***	7	12
11	***15***	7	21	6	11	7	1	3	12
41	***12***	4	6	1	9	27	32	7	5
20	11	3	8	1	51	7	7	***27***	***17***
8	13	5	3	6	6	7	9	***22***	***16***

longitude[scan][pixel] – as filtered by temperature[scan_hi][pixel_hi] w/ dim_multiple

100.3	105.6	106.4	107.8	108.6
100.4	100.7	106.4	105.8	108.6
100.4	105.6	100.5	105.8	108.7
100.6	100.9	100.3	100.6	106.6

Filtering Implementation note...

Under the hood, the plugin analyzes the dimensions of all variables to retrieve. It then builds one or more minimal disjoint sets of dimensions common to these variables. Dimension order is irrelevant: *latitude[line][scan]* and *some_var[scan][band][line]* can both use the common set { line, scan, band }. Each of these sets supplies the dimensions for a ***global filter mask***. Most queries will only have one global filter mask, but we wanted to be sure to handle all possible cases.

The plugin includes a special "multiple-sets of multiple-dimensions" iterator that knows how to iterate across multiple sets of unequal dimensions simultaneously. The iterator is critical because it means we don't have to change the size of any in-memory arrays. (A previous version used the dimension add / remove approach and was much slower in addition to being less flexible).

We apply every filter to the global filter mask(s) by using the multidimensional iterator to walk the filter and the global filter mask simultaneously, while updating the global filter mask according to the logic described above.

To prepare return data, we again use the multidimensional iterator to simultaneously walk the data variable and its global filter mask.

Server-Side Functions

General Purpose Functions

These general-purpose functions were created to help support more complex filtering and averaging use cases. Since they return only standard DAP objects, they may have applicability outside of the MIIC plugin.

Functions Returning Strings

Use these functions anywhere a string is expected.

attr_as_str(*attribute_name*,[string_if_missing])

attr_as_str(*attribute_name*,*array*,[string_if_missing])

Retrieves an attribute value as a string.

The first version uses retrieves the attribute value from the DDS attribute table. This is the main, global table of attributes attached to a file. The second version retrieves an attribute from an attribute table attached to an array.

The optional argument *string_if_missing* will be returned if the attribute does not exist as specified. Otherwise an error is returned.

Return type: string

cat_strs(*string*,...)

Concatenates two or more strings into a single string.

Return type: string

Functions Returning Double

Use these functions anywhere a numeric value (i.e. double) is expected.

str_to_ts(*date_string*,*format*)

Converts a string to a UNIX timestamp.

UNIX timestamp values are created by applying the *format* string to the *date_string* using the C function strptime. Note that you will need to replace the '%' character in *format* with '^', due to HTTP encoding issues.

Return type: double

Functions Returning Arrays

Use these functions anywhere an OPeNDAP array is expected. In the plugin, this is most commonly to provide arrays to filters and data variable functions.

create_array(name,array)

create_array(name,type,dimension_1,...)

Returns a new array with all values initialized to 0 or false.

The first version creates a new array of the same shape and type as the passed-in DAP array. The second version specifies the type and dimensions of the new array.

All DAP primitive types should work in this context:

- Byte
- Char
- Int8
- UInt8
- Int32
- UInt32
- Int64
- UInt64
- Float32
- Float64

Return type: array

Example: Create a new array "flux" with the same type and dimensions as the variable latitude

- create_array("flux",latitude)

clone_array(name,array)

Creates a deep copy of the passed-in array.

Return type: array

Example: Create a new array named "latitude_2" that is a clone of the variable latitude

- create_array("latitude_2",latitude)

fixed_dim(*array*,*dim_name*,*dim_index*)

Creates a new array by constraining one dimension of an array.

This acts like a DAP constraint on an array except that a new array is created and the old array is unchanged. With DAP constraints, you are actually changing the array – and the constraint will be used **everywhere** that array is used, whether you like it or not. As a consequence, this function provides more flexibility but a DAP constraint will probably be faster.

The resulting array is named: miic_fixed_dim_[dim_name]_[dim_index]_[array name].

Return type: array (a constrained clone of the passed-in array)

Example: Here are two ways to choose band 1 from flux_capacity[line][scan][band]

- flux_capacity[0:1:*][0:1:*][1] – *remember, changes every use of flux_capacity*
- fixed_dim(flux_capacity,band,1) – creates a brand new array

gen_time(*min_value*,*max_value*,dimension,...)

Generates time values. This function is a candidate for being replaced with something more general in a future release.

Some files don't include variables for observation time, yet have all the information needed to compute them. This function creates a new time array of UNIX timestamps from min_value to max_value. The size of the array is determined by the provided dimensions. Currently, it simply linearly interpolates time values evenly over all provided dimensions.

Min_value and max_value must be UNIX timestamps (seconds since 1970). These can be strings, numeric values, or arrays. If min_value or max_value are arrays it simply uses the first value in the array.

Return type: DAP array of doubles

Calling Dynamic Expressions

The MIIC plugin supports dynamic expressions which are currently able to access DAP objects and modify DAP arrays.

eval_expr(expr_name,arg0,...,argN)

Evaluate a named expression with the supplied arguments. Returns the first argument (arg0) or NULL.

Expressions are defined in a YAML file stored at the server for security.

The number and types of supplied arguments must agree with the expression definition.

Returns: The first argument (arg0), or NULL if no arguments

Example: Invoke the function colat_to_lat with the CERES colatitude array as the expression argument

```
eval_expr(colat_to_lat,Time_and_Position_Colatitude_of_CERES_FOV_a
t_surface)
```

MIIC Support Functions

These functions return custom DAP Structures, which may be passed as input to other plugin functions.

Functions Returning Dimension Structures

Dimensions wrap OPeNDAP dimension info (name, size, stride, etc). In the MIIC plugin, arrays may also be used in place of functions expecting

Dimensions, in which case the plugin will just extract all dimensions from the array.

get_dim(*array*,*dim_name*)

Selects one dimension from an array.

This is used in functions that require passed-in dimensions.

Return type: dimension info structure

Functions Returning Data Variable Structures

Data Variables define what data is to be included in a tuple or profile.

define_var(*array*)

define_var(*array*,*name*,*lower_bound*|UNBOUND,*upper_bound*|UNBOUND,*missing_val*|UNBOUND)

Includes the specified array for inclusion in a tuple or histogram.

The *array* can come directly from the DDS and may optionally include a projection (e.g. *latitude[0:2:*][0:2:*]* – skip every other value), or you can use any other nested function that returns an array. Note that OPeNDAP projections are applied before any functions are called and are sticky – the projection will apply to all uses of the array, whether you specify the projection again or not.

In the first version, the variable name used in return structures will be the same name as the array.

The second version changes the return name and allows upper, lower and missing values to be provided or set as *UNBOUND*. If these values are provided, they become a local filter, not a global filter. Local filters only exclude values from this data variable, not any other observations that may share the same index.

Local filtering
When returning averaged data, "locally" filtered data is excluded from the bin.

When returning filtered data, "locally" filtered data is returned as the maximum data value from the type of array. We still must return a value because there may be other observations at the same index that are not filtered.

Return type: data variable structure

Functions Returning Filter Structures

Filters may be applied to tuples and histogram functions. These filters are always "global" in that they filter observations from all MIIC Data Variables.

define_filter_mask([DIMS_REQUIRED,]*array*)

Returns a new global filter mask from the passed-in array.

The supplied array is interpreted as a global filter mask. Set values in this array to true or non-zero where you wish to **remove** obse rvations.

The optional DIMS_REQUIRED keyword instructs the filter that it only applies to data variables that have *at least* the same dimensions as the array.

Together with eval_expr, and create_array, this function can be used to create arbitrarily complex filters.

Return type: filter structure

Example: create and apply a new filter mask by evaluating the dynamic expression 'customFilter'. The dimensions of the mask match the dimensions of the latitude variable. The inputs to 'customFilter' are the new mask variable, and variables latitude and longitude.

```
define_filter_mask(eval_expr(mask,"customFilter",create_array(mask
,"BOOLEAN",latitude),latitude,longitude))
```

define_filter_var([DIMS_REQUIRED,]*array*,*min_val*,*max_val*[,data_min,data_max])

Creates a new global range filter. Values less than min_val or larger than max_val will be discarded

The optional DIMS_REQUIRED keyword instructs the filter that it only applies to data variables that have at least the same dimensions as the filter.

An optional data range can be supplied, which must be increasing. The data range permits filters where the range is "inverted". For example, use a longitude range from 170 to -170 (i.e. 20 degrees of longitude):

```
define_filter_var(longitude,170,-170,-180,180)
```

Return type: filter structure

Functions Returning Axis Structures

Axes are used to define the x/y/z histogram and profile axes.

x_axis(*array*,*axis_min*,*axis_max*,*num_bins*,*data_min*,*data_max*)

y_axis(*array*,*axis_min*,*axis_max*,*num_bins*,*data_min*,*data_max*)

z_axis(*array*,*axis_min*,*axis_max*,*num_bins*,*data_min*,*data_max*)

Defines an x, y or z axis.

Values will be "binned" according to the passed in *array.* There will be *num_bins* bins, evenly spaced from *axis_min* to *axis_max* inc lusive. The axis range does not need to be increasing.

Data_min and *data_max* specify the legal range of data values (inclusive) and must be increasing.

Return type: axis structure

Example: create 10 bins from longitude 150 to longitude -150, with a valid longitude range of -180 to 180

```
x_axis(longitude,150,-150,10,-180,180)
```

Functions Customizing Dimension Matching

These functions return Structures which alter how the plugin understands data variable dimensions.

Normally, variable dimensions with the same size and name are considered to match for the purpose of filtering.

"Fake" Dimension Names
For data products where OPeNDAP reports bogus dimension names that start with "fakedim" (e.g. "fakedim33"), only the size needs to match for dimensions to be considered equal. This should probably be replaced with a large dim_alias call.

The CALIPSO data products use fakedim names.

dim_alias(dim_name,alias,...)

Registers aliases for a dimension name.

This function should be used when a data file uses multiple names for the same logical dimension.

Return type: structure

Example: The dimension names "FOV" and "__FOV__" should be treated as matching by the plugin.

```
dim_alias("FOV","__FOV__")
```

dim_multiple(array1,array2)

dim_multiple(dimension_1,...,END,dimension_2,...)

Register one set of dimensions as an even multiple of another set of dimensions.

This function allows the plugin to apply filters to data variables (as described in the filtering section above), where one set of dimensions is an even multiple of another set. This sometimes occurs in data products that provide some data at different resolutions.

The dimensions in first set must be some even multiple N of the dimensions from the second set. The order of dimensions is significant, it will only apply to the same dimensions encountered in the same order.

The first version extracts dimensions from the specified arrays. The second version allows adding individual dimensions. This supports cases where the arrays may have dimensions in addition to the multiplicative dimensions.

Return type: structure

Example: temperature is an even multiple (4) of latitude.

- latitude[1200,1600]
- temperature[2400,3200]

```
dim_multiple(latitude,temperature)
```

Example 2: temperature (without the dimension "band") is an even multiple (4) of latitude.

- latitude[lines=1200,scan=1600]
- temperature[band=7,lines_hi=2400,scan_hi=3200]

```
dim_multiple(get_dim(latitude,"lines"),get_dim(latitude,"scan"),EN
D,get_dim(temperature,"lines_hi"),get_dim(temperature,"scan_hi"))
```

Main Functions

The main functions retrieve filtered or averaged data, usually after applying filters. These are the outermost functions called. At present they may not be nested inside any other functions.

Functions Returning Averaged Data

histo([USE_OVERFLOW,]dim_customize,...,x_axis,filter,...)

histo([USE_OVERFLOW,]dim_customize,...,x_axis,y_axis,filter,...)

histo([USE_OVERFLOW,]dim_customize,...,x_axis,y_axis,z_axis,filter,...)

Returns a 1D, 2D, or 3D histogram.

The provided axes define the histogram as 1D, 2D or 3D. Optional functions may be included to tweak the plugins' understanding of dimensions in your data product.

> Filter objects included here will globally filter all observations. See Filtering above for an explanation of the MIIC multidimensional filtering rules. Note that the shapes of data variables and filters used do not need to match. By default, filters will apply to all data variables that have any dimension(s) in common.

The optional *USE_OVERFLOW* keyword can be used to include overflow and underflow bins in the histogram output. If supplied, values outside of the axis range will also be counted and averaged.

Return type: histogram structure

Example: return a 2d histogram for temperature vs. latitude over the full range. Remove all observations where percent_coverage is >= 50

Variables:

- latitude[FOV]
- longitude[FOV]
- temperature[FOV]
- percent_coverage[FOV]

```
histo(x_axis(longitude,-180,180,10,-180,180),y_axis(latitude,-90,9
0,10,-90,90),define_filter_var(percent_coverage,50,100))
```

profile([USE_OVERFLOW,]dim_customize,...,x_axis,variable,...,filter,...)

profile([USE_OVERFLOW,]dim_customize,...,x_axis,y_axis,variable,...,filter,...)

profile([USE_OVERFLOW,]dim_customize,...,x_axis,y_axis,z_axis,variable,...,filter,...)

Returns a 1D, 2D, or 3D profile with averages, counts, and standard error terms for each requested variable.

The provided axes define the profile as 1D, 2D or 3D. Optional functions may be included to tweak the plugins' understanding of dimensions in your data product.

The axes determine how data variables are "binned" in the output. Note that any data variables must have *at least* the same dimensions as the axis array(s). If they have more dimensions, that's fine -- we'll just include all non-filtered observations in the bins.

Filter objects included here will globally filter all observations. See Filtering above for an explanation of the MIIC multidimensional filtering rules. Note that the shapes of data variables and filters used do not need to match. By default, filters will apply to all data variables that have any dimension(s) in common.

The optional *USE_OVERFLOW* keyword can be used to include overflow and underflow bins in the profile output. If supplied, values outside of the axis range will also be counted and averaged.

Optional functions may be included to tweak the plugins' understanding of dimensions in your data product.

Return type: profile structure

Example: return a 2d profile of temperature using 10 latitude and longitude bins over the entire globe where percent_coverage is >= 50

Variables:

- latitude[FOV]
- longitude[FOV]
- temperature[FOV]
- percent_coverage[FOV]

```
profile(x_axis(longitude,-180,-180,10,-180,180),y_axis(latitude,-9
0,90,10,-90,90),define_var(temperature),define_filter_var(percent_
coverage,50,100))
```

Functions Returning Filtered Observations

compact(dim_customize,...,variable,...,filter,...)

Returns compacted data variables according to the supplied filters.

Filter objects included will globally filter all observations. See Filtering above for an explanation of the MIIC multidimensional filtering rules. Note that the shapes of data variables and filters used do not need to match. By default, filters will apply to all data variables that have any dimension(s) in common.

Optional *dim_customize* functions may be included to tweak the plugins' understanding of dimensions used in the queried data file. This impacts filtering behavior.

Size Warning
If you use very permissive filters or request a lot of data variables, the returned data can become very large. The plugin does not prevent you from returning all data, which would be a lot less efficient than just downloading the whole file.

This function returns compacted arrays. This format is useful for clients that want to exclude arbitrary observations but need to retain existing array shapes.

The plugin returns the smallest possible multidimensional array, using the data type maximum value in place of filtered values where necessary. For example the 4x4 array with pink cells removed can be returned as a compacted 2x3 array:
4x4 array with pink cells filtered (removed)

6	8	11	7
12	6	8	4
11	71	78	21
10	4	53	2

Compacted array size is only 2x3. Maximum value (65535) used for filtered values.

11	65535	65535
10	53	2

In the worst case, compacted arrays could be the same size as an unfiltered array. You might not want to use this function unless you know that your filters are likely to provide some level of compaction of the arrays. For example, the the following filtered array will be the same size as the unfiltered array, despite most of the data being filtered (the pink cells):

	🟥	🟥	
🟥	🟥		🟥
🟥		🟥	🟥
	🟥		🟥

Return type: compact array structure

Example: return temperature in the northern latitudes where percent_coverage is >= 90

Variables:

- latitude[FOV]
- temperature[FOV]
- percent_coverage[FOV]

```
compact(define_var(temperature),define_filter_var(latitude,0,90),d
efine_filter_var(percent_coverage,90,100))
```

tuple([NO_TUPLE_MERGE,]dim_customize,...,variable,...,filter,...)

Returns a tuple (a 1D list of observations) with data values from each requested variable, using the requested filters.

Optional *dim_customize* functions may be included to tweak the plugins' understanding of dimensions used in the queried data file. This impacts filtering behavior.

Filter objects included here will globally filter all observations. See Filtering above for an explanation of the MIIC multidimensional filtering rules. Note that the shapes of data variables and filters used do not need to match. By default, filters will apply to all data variables that have any dimension(s) in common.

This function will, by default, filter the requested data variables and return them as one single *tuple* definition in output. This "tuple merging" behavior is convenient for clients who want to analyze data according to logical "observations".

For example, when returning the data variables *latitude[line][scan]*, *longitude[line][scan]* and *temperature[band][line][scan],* the function returns combined observations that pass the filters:

- Observation[band=0][line=7][scan=12]: latitude=33.3, longitude=101.337, temperature=42.1
- Observation[band=1][line=7][scan=12]: latitude=33.3, longitude=101.337, temperature=33.4
- Observation[band=0][line=7][scan=14]: latitude=33.301, longitude=101.653, temperature=42.55
- ...

If the dimension merging behavior is not desired, use the keyword **NO_TUPLE_MERGE**. This instructs the plugin to return a separate 1D tuple for each unique data variable shape. In the above example two tuples will be returned: one for latitude/longitude, and one for temperature.

Size Warning
If you have very permissive filters, or request a lot of data variables, the tuples can become very large.

Also note that default tuple merging (absent the **NO_TUPLE_MERGE** keyword) can result in very large tuples if you request variables of different shapes.

Return type: tuple structure

Example: return a tuple containing temperature in the northern latitudes where percent_coverage is >= 90

Variables:

- latitude[FOV]
- temperature[FOV]
- percent_coverage[FOV]

```
tuple(define_var(temperature),define_filter_var(latitude,0,90),def
ine_filter_var(percent_coverage,90,100))
```

Dynamic Expressions

The MIIC plugin integrates dynamic expressions evaluated by the "exprtk" library (http://www.partow.net/programming/exprtk/). This is a C++ header-only library and does not invoke any sort of external toolchain which could potentially be exploited. Trusted users can provide their own mathematical formulas or algorithms to process data on the server. This provides the advantage of not having to compile, link and deploy a new MIIC plugin every time an algorithm changes. Expressions are evaluated for their side-effects – they are able to change the content of DAP arrays.

Performance of dynamic expressions is very good. For mathematical operations and loops, performance is typically the same as compiled code. Boolean logic performance is worse, being about twice as slow as compiled code.

Dynamic expressions in MIIC use good security practices. The expressions themselves are defined in a YAML text file on the server so they cannot be modified by users. Instead, users call expressions by name, providing other DAP objects as input. This is accomplished with the eval_expr function.

ExprTk Capabilities

Expressions are written in a C-like language. From the ExprtTK website:

```
The ExprTk expression evaluator supports the following fundamental
arithmetic operations, functions and processes:
 (00) Types:           Scalar, Vector, String
 (01) Basic operators: +, -, *, /, %, ^
 (02) Assignment:      :=, +=, -=, *=, /=, %=
 (03) Equalities &
      Inequalities:    =, ==, <>, !=, <, <=, >, >=
 (04) Boolean logic:   and, mand, mor, nand, nor, not, or, shl, shr,
                       xnor, xor, true, false
 (05) Functions:       abs, avg, ceil, clamp, equal, erf, erfc,  exp,
                       expm1, floor, frac,  log, log10, log1p,  log2,
                       logn,  max,  min,  mul,  ncdf,  nequal,  root,
                       round, roundn, sgn, sqrt, sum, swap, trunc
 (06) Trigonometry:    acos, acosh, asin, asinh, atan, atanh,  atan2,
                       cos,  cosh, cot,  csc, sec,  sin, sinc,  sinh,
                       tan, tanh, hypot, rad2deg, deg2grad,  deg2rad,
                       grad2deg
 (07) Control
      structures:      if-then-else, ternary conditional, switch-case,
                       return-statement
 (08) Loop statements: while, for, repeat-until, break, continue
 (09) String
      processing:      in, like, ilike, concatenation
 (10) Optimisations:   constant-folding, simple strength reduction and
                       dead code elimination
 (11) Calculus:        numerical integration and differentiation
```

Adding Dynamic Expressions

The MIIC plugin loads expressions from YAML definition file(s) stored on the server, listed in the MIIC module configuration file. The config file is typically located at */etc/bes/modules/miic.conf*.

For example, this MIIC configuration will load dynamic expressions from two separate locations:

```
# MIIC expression file
MIIC.ExpressionFile=/etc/bes/modules/miic_expressions.yaml;/home/currey/te
st_expressions.yaml
```

The YAML configuration file(s) contains a list of expressions. Expressions are uniquely identified by name and can accept one or more objects (DAP arrays, strings, or numeric values) as input. The types of inputs must be provided as a formal argument list. Arguments are named sequentially (arg0, arg1, arg2, etc.) and the supported types are:

- **string**: for DAP strings
- **scalar**: for all DAP numeric types (values are always promoted to double)
- **vector**: for DAP arrays (only arrays of primitive numeric types are supported)

The following example shows the YAML configuration syntax of the expression "colat_to_lat". Note that it only has one argument: "arg0" – a vector that is expected to contain colatitude values:

```
- name: colat_to_lat
   description: "convert array of colatitude to latitude"
   args: [vector]
   expression: >
     for (var i:=0; i < arg0_length; i+=1) {
       var colat:= arg0(i);
       if (colat <= 180) {
         set_arg0(i,90-colat);
       }
     }
```

Arguments are available for use inside the expression:

- **String** and **scalar** arguments are available directly as the symbol ***argN*** as in "var a:=arg3+27;"
- **Vector** arguments provide symbols for the size & shape of DAP arrays, plus functions to get & set values:
  - **arg*N*_shape** -- a vector containing the shape of the DAP array. For example, "arg3_shape[]" will be the number of dimensions, "arg3_shape[0]" is the length of the first dimension, etc.
  - ***argN_length*** -- a constant symbol containing the length of the DAP array. For example, "for (var i:=0; i<arg1_length; i++) { ... }" will loop over all elements in a DAP array passed in as the 2nd argument (arg1).
  - **arg*N*(i)** -- a function to return the value at index i. The return value is always promoted to a double.
  - ***set_argN*(i, val)** -- a function to change the value at index i.

Expressions may be invoked by calling the eval_expr function.

Data Formats

Histograms

The histogram data format begins with a DAP structure named "histogram_data". This structure contains arrays defining the histogram bins and arrays containing counts, means, and standard errors for each axis.

The following arrays definine the upper and lower bounds of each axis bin. The size of these arrays is always the # of bins requested for the corresponding axis:

- x_LOW – an integer array of the lower-bounds of each x axis bin (1D, 2D & 3D histograms)
- x_HIGH – an integer array of the upper-bounds of each x axis bin (1D, 2D & 3D histograms)
- y_LOW – an integer array of the lower-bounds of each y axis bin (2D & 3D histograms)
- y_HIGH – an integer array of the upper-bounds of each y axis bin (2D & 3D histograms)
- z_LOW – an integer array of the lower-bounds of each z axis bin (3D histograms)
- z_HIGH – an integer array of the upper-bounds of each z axis bin (3D histograms)

These arrays contains the histogram statistics:

- COUNT – an integer array of counts (1D, 2D & 3D histograms)
- x_MEAN – a double array of averaged axis values (1D, 2D & 3D histograms)
- x_STD – a double array of standard error terms (1D, 2D & 3D histograms)
- y_MEAN – a double array of averaged axis values (2D & 3D histograms)
- y_STD – a double array of standard error terms (2D & 3D histograms)
- z_MEAN – a double array of averaged axis values (3D histograms)
- z_STD – a double array of standard error terms (3D histograms)

Note that the statistics arrays returned have the following dimensions. If using overflow bins, the size of each dimension is increased by two to accommodate **overflow** and **underflow** values. In this case, values at index 0 will always be **underflow** and the value at index (size-1) will be **ov erflow**.

- 1D histograms: [xbins]
- 2D histograms: [xbins][ybins] – y changes fastest

- 3D histograms: [xbins][ybins][zbins] – z changes fastest

Example:

longitude[line][scan]

100	100	101
100	100	101
100	100	101

latitude[line][scan]

27	27	27
30	30	30
33	34	34

Return a 2D histogram with X axis as 4 longitude bins from 100 to 104, Y axis as 4 latitude bins from 30 to 34, no overflow bins, and no additional filtering:

```
yourfile.nc?histo(x_axis(longitude,100,104,4,-180,180),y_axis(latitude,30,
34,4,-90,90))
```

Return structure:

histogram_data.x_LOW = { 100, 101, 102, 103 }

histogram_data.x_HIGH = { 101, 102, 103, 104 }

histogram_data.y_LOW = { 30, 31, 32, 33 }

histogram_data.y_HIGH = { 31, 32, 33, 34 }

histogram_data.COUNT = { { 2, 0, 0, 2 }, { 1, 0, 0, 1 }, {0, 0, 0, 0 }, {0, 0, 0, 0 } }

histogram_data.x_MEAN = { {100, 0, 0, 100 }, { 0, 0, 0, 0 }, {0, 0, 0, 0 }, {0, 0, 0, 0 } }

histogram_data.x_STD = { { 0, 0, 0, 0 }, { 0, 0, 0, 0 }, {0, 0, 0, 0 }, {0, 0, 0, 0 } }

histogram_data.y_MEAN = { { 30, 0, 0, 33.5 }, { 30, 0, 0, 34 }, {0, 0, 0, 0 }, {0, 0, 0, 0 } }

histogram_data.y_STD = { { 0, 0, 0, 0.5 }, { 0, 0, 0, 0 }, {0, 0, 0, 0 }, {0, 0, 0, 0 } }

Profiles

The profile data format is a structure named "profile_data". This structure contains arrays defining the profile bins and arrays containing counts, averaged data variables, and standard error.

The following arrays definine the upper and lower bounds of each axis bin. The size of these arrays is always the # of bins requested for the corresponding axis:

- x_LOW – an integer array of the lower-bounds of each x axis bin (1D, 2D & 3D profiles)
- x_HIGH – an integer array of the upper-bounds of each x axis bin (1D, 2D & 3D profiles)
- y_LOW – an integer array of the lower-bounds of each y axis bin (2D & 3D profiles)
- y_HIGH – an integer array of the upper-bounds of each y axis bin (2D & 3D profiles)
- z_LOW – an integer array of the lower-bounds of each z axis bin (3D profiles)
- z_HIGH – an integer array of the upper-bounds of each z axis bin (3D profiles)

Each requested data variable returns three separate statistics arrays:

- [variable name]_COUNT – an integer array of counts.

- [variable name]_MEAN – a double array of averages.
- [variable name]_STD – a double array of standard error terms.

Note that the statistics arrays returned have the following dimensions. If using overflow bins, the size of each dimension is increased by two to accommodate **overflow** and **underflow** values. In this case, values at index 0 will always be **underflow** and the value at index (size-1) will be **overflow**.

- 1D profiles: [xbins]
- 2D profiles: [xbins][ybins] – y changes fastest
- 3D profiles: [xbins][ybins][zbins] – z changes fastest

Example:
flux[line][scan]

12	32	31	11	1
15	12	17	21	41
24	27	26	21	33
33	44	32	31	11
7	41	17	16	31

latitude[line][scan]

33.1	33.2	33.1	33.1	33.2
34.1	34.1	34.2	34.2	34.1
34.8	34.8	34.9	34.8	34.9
35.3	35.4	35.3	35.4	35.3
36.0	36.0	35.9	36.0	36.0

longitude[line,scan]

100.1	100.9	101.5	102.2	102.8
100.2	100.9	101.5	102.2	102.8
100.1	100.8	101.4	102.1	102.7
100	100.9	100.4	102.1	102.8
100	101	100.5	102.2	102.8

Create profile from variable flux, with X axis as 4 longitude bins from 30 to 50, Y axis as 4 latitude bins from 100 to 120, no overflow bins:

```
yourfile.nc?profile(x_axis(longitude,30,50,4,-180,180),y_axis(latitude,100
,120,4,-90,90),define_var(flux))
```

Return structure:

DIM xbins=4

DIM ybins=4

profile_data.x_LOW[xbins] = { 100, 105, 110, 115 }

profile_data.x_HIGH[xbins] = { 105, 110, 115, 120 }

profile_data.y_LOW[ybins] = { 30, 35, 40, 45 }

profile_data.y_HIGH[ybins] = { 35, 40, 45, 50 }

profile_data.flux_COUNT[xbins][ybins] = { { 15, 10, 0, 0 }, { 0, 0, 0, 0 }, {0, 0, 0, 0 }, {0, 0, 0, 0 } }

profile_data.flux_MEAN[xbins][ybins] = { { 21.6, 26.3, 0, 0 }, { 0, 0, 0, 0 }, {0, 0, 0, 0 }, {0, 0, 0, 0 } }

profile_data.flux_STD[xbins][ybins] = { { 10.50034, 12.69339, 0, 0 }, { 0, 0, 0, 0 }, {0, 0, 0, 0 }, {0, 0, 0, 0 } }

Compact Arrays

This data format preserves the shape of data arrays.

Arrays are returned in a structure named "compact_data" containing groups for each unique array "shape" returned (shape_0, shape_1, etc.)

Inside shape-groups are arrays for the requested data variables sharing the same shape. These arrays have the same dimensions as the original data variables. However, they may be "compacted" such that indices or dimensions containing only filtered values are removed. The data types' maximum value is used to indicate filtered values.

Dimensions that could not be compacted will have the same name and size as the original data variable. When a dimension is compacted (i.e. some indices are removed) the dimension name is changed to *shape_n_[original dimension name]_compact*, and the size will be something less than the original size.

Each shape-group also includes arrays containing the original indices for the "compacted" dimensions. We add this so clients can know precisely what data was returned. This array name is *shape_n_[original dimension name]_indices*. It contains a 1D list of the indices that are present in the output data.

Example: The filtered structure for data variable *latitude[line][scan]*, where the pink cells are filtered. There is only one shape returned (shape_0).

Note that the *scan* dimension was compacted and renamed *scan_compact*, but the line dimension is unchanged. Because the scan dimension was compacted, we also have an array *scan_indices.* This array lets users know that data from the "scan_compact" dimension came from scan=1 and scan=2.

latitude[line][scan]

55.2	55.05	54.4
56.1	55.12	55.43
55.7	55.703	55.1

DIM line=3

DIM shape_0_scan_compact=2

compact_data.shape_0.latitude[line][shape_0_scan_compact] = { { 55.05, 54.4 }, {55.12, 55.43 }, { 55.703, 1.79769e+308 } }

compact_data.shape_0.shape_0_scan_indices[shape_0_scan_compact] = { 1, 2 }

Tuples

The tuple format is returned as a structure is named "tuple_data" and contains groups for one or more tuples (tuple_0, tuple_1, etc.). Each tuple group defines one index array and one or more data variable arrays. Note that by default, only one tuple (tuple_0) will be returned.

Tuple groups have an array for each data variable sharing the tuple definition. This is a 1D array, where the only dimension is the tuple dimension. This array contains your "tupleized" data.

Tuple groups also have an index array. This array lists the original indices of each tuple value, allowing clients to know exactly what data was returned in the tuple. The index array is named "tuple_0_indices", for tuple_0, and so on. This array has two dimensions:

- tuple_0, tuple_1, etc: This is the tuple dimension, i.e. the observations returned
- tuple_0_dims, tuple_1_dims, etc: The total number of dimensions from all data variables returned in this tuple

---

Example: The tuple structure for data variable *latitude[line][scan]*, where the pink cells are filtered:

55.2	55.05	54.4
56.1	55.12	55.43
55.7	55.703	55.1

Return structure:

DIM tuple_0=5

DIM tuple_0_dims=2 (i.e. line and scan)

tuple_data.tuple_0.latitude[tuple_0] = { 55.05, 54.4, 55.12, 55.43, 55.703 }

tuple_data.tuple_0.tuple_0_indices[tuple_0][tuple_0_dims] = { { 0, 1 }, { 0, 2 }, { 1, 1 }, { 1, 2 }, { 2, 1 } }

Additional Examples

The following examples use the test data that ships with the OPeNDAP Hyrax server.

Example 1

Using data file S2000415.HDF.gz.

Filter wind speed where latitude is between 0 and 30 degrees:

```
https://[server URL]/opendap/data/hdf4/S2000415.HDF.gz.ascii?tuple(define_filter_var(NSCAT_Rev_20_WVC_Lat,0,30),define_var(NSCAT_Rev_20_Wind_Speed))
```

Same filter, but with multi-dimensional output:

```
https://[server URL]/opendap/data/hdf4/S2000415.HDF.gz.ascii?compact(define_filter_var(NSCAT_Rev_20_WVC_Lat,0,30),define_var(NSCAT_Rev_20_Wind_Speed))
```

Bin wind speed by latitude, and also filter between 0 and 30 degrees latitude:

```
https://[server URL]/opendap/data/hdf4/S2000415.HDF.gz.ascii?histo(x_axis(NSCAT_Rev_20_WVC_Lat,0,30,10,-90,90),define_filter_var(NSCAT_Rev_20_WVC_Lat,0,30),define_var(NSCAT_Rev_20_Wind_Speed))
```